INGENIERÍA DE APLICACIONES

Patrones de Diseño

Dra. María Luján Ganuza
mlg@cs.uns.edu.ar
DCIC - Depto. de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur, Bahía Blanca
2019

Temario

Patrones de Diseño

Patrones Creacionales

- Abstract Factory
- Prototype

Patrones Estructurales

- Adapter
- Composite

Patrones de Comportamiento

- Command
- Observer

Software Reusable

La **reutilización** de Software ofrece confiabilidad, menor tiempo de desarrollo, menores costos, etc.

Existen varios grados de reutilización: **de código**, **de módulos, de ideas de diseño**.

Los **mecanismos técnicos** que facilitan la reutilización son: clases, herencia, composición de objetos, genericidad, etc.

Alcanzar un buen grado de reusabilidad no es fácil:
queremos beneficiarnos de la reusabilidad ahora.
debemos trabajar ahora para beneficiarnos en el futuro.

Software Reusable

- En general, muchas de las dificultades en alcanzar un buen programa radican en el **diseño**.
- En orientación a objetos, el buen diseño no es fácil de conseguir.
  - Siempre pensando en la **reutilización del software**,
    debemos **encontrar los objetos** pertinentes,
    factorizarlos en clases con una granularidad adecuada,
    definir **interfaces**,
    jerarquías de herencia,
    establecer **relaciones** entre clases... etc.
- El diseño debe ser adecuado para la aplicación en particular.
- El diseño debe ser lo mas general y flexible posible.

Software Reusable

- Los programadores y desarrolladores experimentados no resuelven las cosas desde cero. ***Reutilizan*** *soluciones previamente encontradas, testeadas y aprobadas.*
    - Lo hacen en el **código** con facilidad al identificar el procedimiento a implementar.
    - Lo hacen en el **diseño** con facilidad al identificar la aplicación particular.

Básicamente, ***identifican patrones*** *y aplican soluciones.*

Patrones de Diseño

Los patrones de diseño **nombran**, **explican** y **evalúan** un diseño importante y recurrente en los sistemas orientados a objetos.

En general, poseen cuatro elementos:

Nombre del patrón: para poder identificarlo fácilmente y tratarlo de manera abstracta cuando sea necesario.

El problema: que describe cuándo utilizar este patrón de forma tal que sea la solución.

La solución: que describe los elementos que componen el diseño, sus relaciones, sus responsabilidades, etc. Se describe en forma abstracta.

Las consecuencias: que son los resultados y el balance final de aplicar el patrón impacto en el sistema, detalles de lenguajes, etc)

Patrones de Diseño

Los **patrones de diseño** son básicamente descripciones de objetos que se comunican y clases que son personalizadas para resolver un **problema de diseño** general en un contexto particular.

Los patrones se describen **gráficamente**, lo que facilita su comprensión, pero no es suficiente.

Algunos aspectos no pueden ser aclarados o especificados por medio de diagramas.

Se adopta entonces una convención para la descripción de patrones.

Es un formato generalmente aceptado que incluye todos los *items* a destacar.

Descripción de Patrones de Diseño

- **Nombre del patrón** y clasificación
- **Intención**
- **Alias**
- **Motivación**
- **Aplicabilidad**
- **Estructura**
- **Participantes**
- **Colaboraciones**
- **Consecuencias**
- **Implementación**
- **Código ejemplo**
- **Usos conocidos**
- **Patrones relacionados**

Descripción de Patrones de Diseño

- **Nombre del patrón**
- **Intención**
  ¿qué hace? ¿qué problema ataca?
- **Alias**
- **Motivación**
- **Aplicabilidad**
- **Estructura**
- **Participantes**
- **Colaboraciones**
- **Consecuencias**
- **Implementación**
- **Código ejemplo**
- **Usos conocidos**
- **Patrones relacionados**

Descripción de Patrones de Diseño

- **Nombre del patrón**
- **Intención**
- **Alias**
  algunos patrones son conocidos por nombres diferentes.
- **Motivación**
- **Aplicabilidad**
- **Estructura**
- **Participantes**
- **Colaboraciones**
- **Consecuencias**
- **Implementación**
- **Código ejemplo**
- **Usos conocidos**
- **Patrones relacionados**

Descripción de Patrones de Diseño

- **Nombre del patrón**
- **Intención**
- **Alias**
- **Motivación:** un escenario que describe el problema de diseño y que muestra como el patrón resuelve el problema.
- **Aplicabilidad**
- **Estructura**
- **Participantes**
- **Colaboraciones**
- **Consecuencias**
- **Implementación**
- **Código ejemplo**
- **Usos conocidos**
- **Patrones relacionados**

Descripción de Patrones de Diseño

- **Nombre del patrón**
- **Intención**
- **Alias**
- **Motivación**
- **Aplicabilidad**
  cuáles son las situaciones en las cuales se aplica el patrón.
- **Estructura**
- **Participantes**
- **Colaboraciones**
- **Consecuencias**
- **Implementación**
- **Código ejemplo**
- **Usos conocidos**
- **Patrones relacionados**

Descripción de Patrones de Diseño

- **Nombre del patrón**
- **Intención**
- **Alias**
- **Motivación**
- **Aplicabilidad**
- **Estructura:** representación gráfica de las clases del patrón (diagramas de clases, diagramas de interacción)
- **Participantes**
- **Colaboraciones**
- **Consecuencias**
- **Implementación**
- **Código ejemplo**
- **Usos conocidos**
- **Patrones relacionados**

Descripción de Patrones de Diseño

- **Nombre del patrón**
- **Intención**
- **Alias**
- **Motivación**
- **Aplicabilidad**
- **Estructura**
- **Participantes:** clases y objetos que participan del patrón.
- **Colaboraciones**
- **Consecuencias**
- **Implementación**
- **Código ejemplo**
- **Usos conocidos**
- **Patrones relacionados**

Descripción de Patrones de Diseño

- **Nombre del patrón**
- **Intención**
- **Alias**
- **Motivación**
- **Aplicabilidad**
- **Estructura**
- **Participantes**
- **Colaboraciones**
  cómo los participantes colaboran para realizar alguna tarea.
- **Consecuencias**
- **Implementación**
- **Código ejemplo**
- **Usos conocidos**
- **Patrones relacionados**

Descripción de Patrones de Diseño

- **Nombre del patrón**
- **Intención**
- **Alias**
- **Motivación**
- **Aplicabilidad**
- **Estructura**
- **Participantes**
- **Colaboraciones**
- **Consecuencias**
  beneficios, balance pros y contras, etc.
- **Implementación**
- **Código ejemplo**
- **Usos conocidos**
- **Patrones relacionados**

Descripción de Patrones de Diseño

- **Nombre del patrón**
- **Intención**
- **Alias**
- **Motivación**
- **Aplicabilidad**
- **Estructura**
- **Participantes**
- **Colaboraciones**
- **Consecuencias**
- **Implementación**
  hints, técnicas y detalles a tener en cuenta al implementar el patrón.
- **Código ejemplo**
- **Usos conocidos**
- **Patrones relacionados**

Descripción de Patrones de Diseño

- **Nombre del patrón**
- **Intención**
- **Alias**
- **Motivación**
- **Aplicabilidad**
- **Estructura**
- **Participantes**
- **Colaboraciones**
- **Consecuencias**
- **Implementación**
- **Código ejemplo**
  fragmentos de código que ejemplifican la implementación
- **Usos conocidos**
- **Patrones relacionados**

Descripción de Patrones de Diseño

- **Nombre del patrón**
- **Intención**
- **Alias**
- **Motivación**
- **Aplicabilidad**
- **Estructura**
- **Participantes**
- **Colaboraciones**
- **Consecuencias**
- **Implementación**
- **Código ejemplo**
- **Usos conocidos**
  ejemplos de patrones en sistemas reales.
- **Patrones relacionados**

Descripción de Patrones de Diseño

- **Nombre del patrón**
- **Intención**
- **Alias**
- **Motivación**
- **Aplicabilidad**
- **Estructura**
- **Participantes**
- **Colaboraciones**
- **Consecuencias**
- **Implementación**
- **Código ejemplo**
- **Usos conocidos**
- **Patrones relacionados**
  qué otros patrones de diseño están relacionados o pueden combinarse para resolver problemas mayores.

Patrones de Diseño

El catálogo de Patrones de Diseño está compuesto por 23 patrones:

1. Abstract Factory
2. Adapter
3. Bridge
4. Builder
5. Chain of Responsibility
6. Command
7. Composite
8. Decorator
9. Facade
10. Factory Method
11. Flyweight
12. Interpreter
13. Iterator
14. Mediator
15. Memento
16. Observer
17. Prototype
18. Proxy
19. Singleton
20. State
21. Strategy
22. Template Method
23. Visitor

Patrones de Diseño

El catálogo de Patrones de Diseño está compuesto por 23 patrones:

1. **Abstract Factory**
2. **Adapter**
3. Bridge
4. Builder
5. Chain of Responsibility
6. **Command**
7. **Composite**
8. Decorator
9. Facade
10. Factory Method
11. Flyweight
12. Interpreter
13. Iterator
14. Mediator
15. Memento
16. **Observer**
17. **Prototype**
18. Proxy
19. Singleton
20. State
21. Strategy
22. Template Method
23. Visitor

Patrones de Diseño

Los patrones de diseño varían en su granularidad y nivel de abstracción.

Como hay muchos patrones de diseño, necesitamos una forma de organizarlos.

Los patrones se clasifican en base a dos criterios:

- **Propósito**: refleja qué hace el patrón
  - **Creacional**: conciernen al proceso de creación de objetos.
  - **Estructural:** tratan la composición de clases u objetos.
  - **de Comportamiento:** caracterizan las formas en que las clases u objetos interactúan y distribuyen la responsabilidad
- **Alcance**: especifica si el patrón se aplica primariamente a **clases** o a **objetos**.

Patrones de Diseño

		PROPÓSITO		
		CREACIONAL	ESTRUCTURAL	COMPORTAMIENTO
ALCANCE	CLASE	Factory Method	Adapter	Interpreter Template Method
	OBJETO	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

PATRONES CREACIONALES

Creational Patterns

Patrones Creacionales

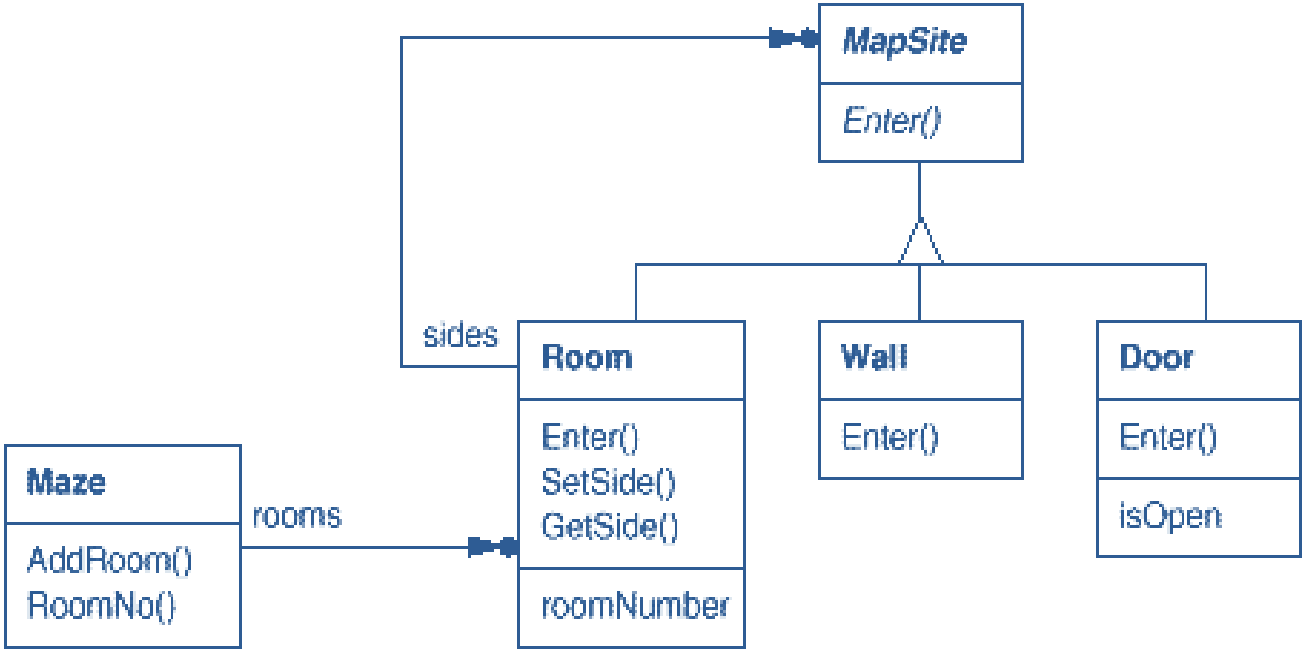
Los **patrones creacionales** son patrones que abstraen el proceso de instanciación.

Procuran independizar el sistema de cómo sus objetos son **creados**, **compuestos** y **representados**.

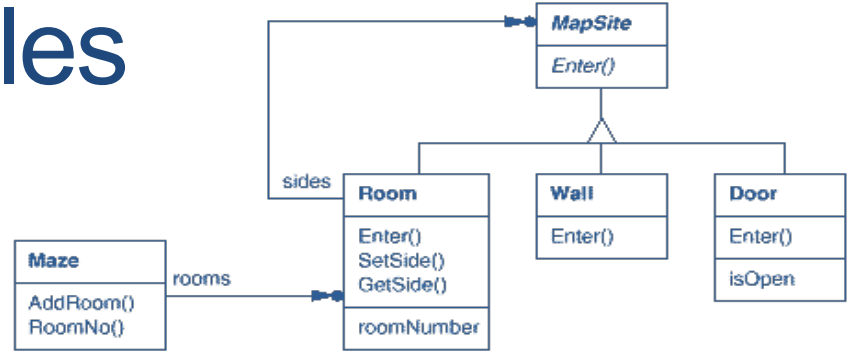
Los patrones indican soluciones de diseño para **encapsular** el conocimiento acerca de las clases que el sistema usa y **ocultar** cómo se crean instancias de estas clases.

Patrones Creacionales

Un laberinto (maze) es un conjunto de habitaciones (rooms). Una habitación conoce a sus vecinos, los cuales pueden ser otra habitación, un muro o una puerta a otra habitación.



Patrones Creacionales



- Cada habitación tiene cuatro lados. En C++ podemos declarar un enumerado:

```
enum Direction {North, South, East, West};
```

- La clase **MapSite** es una clase abstracta para todos los componentes del laberinto. Posee una sola operación **Enter()** para simplificar.
- El significado de **Enter()** depende del componente: *Si es una habitación, cambiamos de locación, si es una puerta y está abierta, pasamos a la siguiente habitación.*

Patrones Creacionales

```
class Room : public MapSite {
public:
        Room(int roomNo);
        MapSite* GetSide(Direction) const;
        void SetSide(Direction, MapSite*);
        virtual void Enter();
private:
        MapSite* _sides[4];
        int _roomNumber;
};
```

```
class Door : public MapSite {
public:
  Door(Room* = 0, Room* = 0);
  virtual void Enter();
  Room* OtherSideFrom(Room*);
private:
  Room* _room1; Room* _room2;
  bool _isOpen; };
```

```
class Wall:public MapSite
{
public:
  Wall();
  virtual void Enter();
};
```

```
class Maze {
public:
    Maze();
    void AddRoom(Room*);
    Room* RoomNo(int)const;
private:
        // ...
};
```

Patrones Creacionales

En la clase **MazeGame** necesitamos crear un laberinto para el juego....

```
Maze* MazeGame::CreateMaze () {
        Maze* aMaze = new Maze;
        Room* r1 = new Room(1);
        Room* r2 = new Room(2);
        Door* theDoor = new Door(r1, r2);
        aMaze->AddRoom(r1);
        aMaze->AddRoom(r2);
        r1->SetSide(North, new Wall());
        r1->SetSide(East, theDoor);
        r1->SetSide(South, new Wall());
        r1->SetSide(West, new Wall());
        r2->SetSide(North, new Wall());
        r2->SetSide(East, new Wall());
        r2->SetSide(South, new Wall());
        r2->SetSide(West, theDoor);
        return aMaze;
}
```

Patrones Creacionales

En la clase **MazeGame** necesitamos crear un laberinto para el juego....

```
Maze* MazeGame::CreateMaze () {
        Maze* aMaze = new Maze;
        Room* r1 = new Room(1);
        Room* r2 = new Room(2);
        Door* theDoor = new Door(r1, r2);
        aMaze->AddRoom(r1);
        aMaze->AddRoom(r2);
        r1->SetSide(North, new Wall());
        r1->SetSide(East, theDoor);
        r1->SetSide(South, new Wall());
        r1->SetSide(West, new Wall());
        r2->SetSide(North, new Wall());
        r2->SetSide(East, new Wall());
        r2->SetSide(South, new Wall());
        r2->SetSide(West, theDoor);
        return aMaze;
}
```

Patrones Creacionales

En la clase **MazeGame** necesitamos crear un laberinto para el juego....

```
Maze* MazeGame::CreateMaze () {
        Maze* aMaze = new Maze;
        Room* r1 = new Room(1);
        Room* r2 = new Room(2);
        Door* theDoor = new Door(r1, r2);
        aMaze->AddRoom(r1);
        aMaze->AddRoom(r2);
        r1->SetSide(North, new Wall());
        r1->SetSide(East, theDoor);
        r1->SetSide(South, new Wall());
        r1->SetSide(West, new Wall());
        r2->SetSide(North, new Wall());
        r2->SetSide(East, new Wall());
        r2->SetSide(South, new Wall());
        r2->SetSide(West, theDoor);
        return aMaze;
}
```

Patrones Creacionales

En la clase **MazeGame** necesitamos crear un laberinto para el juego....

```
Maze* MazeGame::CreateMaze () {
        Maze* aMaze = new Maze;
        Room* r1 = new Room(1);
        Room* r2 = new Room(2);
        Door* theDoor = new Door(r1, r2);
        aMaze->AddRoom(r1);
        aMaze->AddRoom(r2);
        r1->SetSide(North, new Wall());
        r1->SetSide(East, theDoor);
        r1->SetSide(South, new Wall());
        r1->SetSide(West, new Wall());
        r2->SetSide(North, new Wall());
        r2->SetSide(East, new Wall());
        r2->SetSide(South, new Wall());
        r2->SetSide(West, theDoor);
        return aMaze;
}
```

Esta operación crea un laberinto con dos habitaciones

Puede simplificarse. Por ejemplo, las habitaciones podrían crear las paredes.

Pero esto sólo mueve código de un lugar a otro ☹

Patrones Creacionales

En la clase **MazeGame** necesitamos crear un laberinto para el juego....

```
Maze* MazeGame::CreateMaze () {
        Maze* aMaze = new Maze;
        Room* r1 = new Room(1);
        Room* r2 = new Room(2);
        Door* theDoor = new Door(r1, r2);
        aMaze->AddRoom(r1);
        aMaze->AddRoom(r2);
        r1->SetSide(North, new Wall());
        r1->SetSide(East, theDoor);
        r1->SetSide(South, new Wall());
        r1->SetSide(West, new Wall();
        r2->SetSide(North, new Wall());
        r2->SetSide(East, new Wall());
        r2->SetSide(South, new Wall());
        r2->SetSide(West, theDoor);
        return aMaze;
}
```

Además **¿Qué pasa si queremos agregar otro elemento al laberinto o modificar uno existente?**

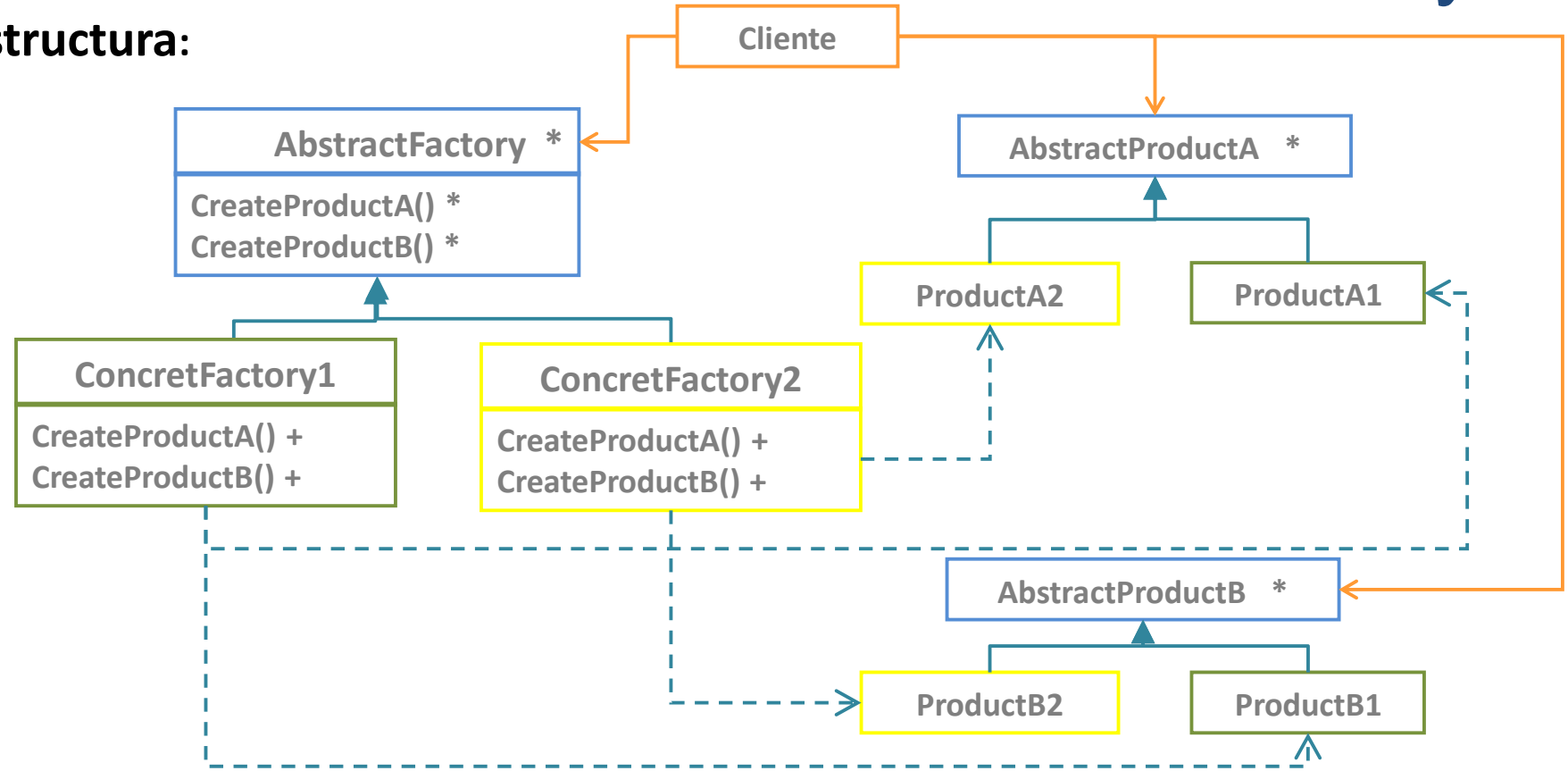
Los patrones creacionales procuran hacer este diseño más flexible, quitando las referencias explícitas a clases concretas.

Patrones Creacionales: Abstract Factory

- **Intención: Proveer una interfaz para crear familias de objetos dependientes o relacionados sin especificar sus clases concretas.**
- **Alias**: Kit (como sufijo usualmente).
- **Aplicabilidad**: Usamos este patrón cuando
  - Un sistema debe independizarse de cómo sus productos son creados, compuestos y representados.
  - Un sistema debe ser configurado con una de múltiples familias de productos.
  - Una familia de objetos debe ser usada en conjunto, y debe reforzarse este hecho.
  - Queremos proveer una librería de productos, pero sólo publicitar las interfaces, no las implementaciones.

Patrones Creacionales: Abstract Factory

Estructura:



Patrones Creacionales: Abstract Factory

Participantes:

- **AbstractFactory:** declara una interfaz para operaciones que crean objetos producto (abstractos)
- **ConcreteFactory**: implementa las operaciones para crear objetos producto concretos
- **AbstractProduct:** declara una interfaz para un tipo de producto
- **ConcreteProduct**: define un objeto producto que será creado por la correspondiente clase factory concreta. Implementa la interfaz *AbstractProduct*.
- **Cliente**: usa sólo interfaces declaradas por *AbstractFactory* y *AbstractProduct*.

Patrones Creacionales: Abstract Factory

Apliquemos el patrón ***Abstract Factory*** a los laberintos…

```
class MazeFactory {
   public:
      MazeFactory();
      virtual Maze* MakeMaze() const
            { return new Maze; }
      virtual Wall* MakeWall() const
            { return new Wall; }
      virtual Room* MakeRoom(int n) const
            { return new Room(n); }
      virtual Door* MakeDoor(Room* r1, Room* r2) const
            { return new Door(r1, r2); }
};
```

La clase **MazeFactory** crea componentes clásicos de laberinto.

Patrones Creacionales: Abstract Factory

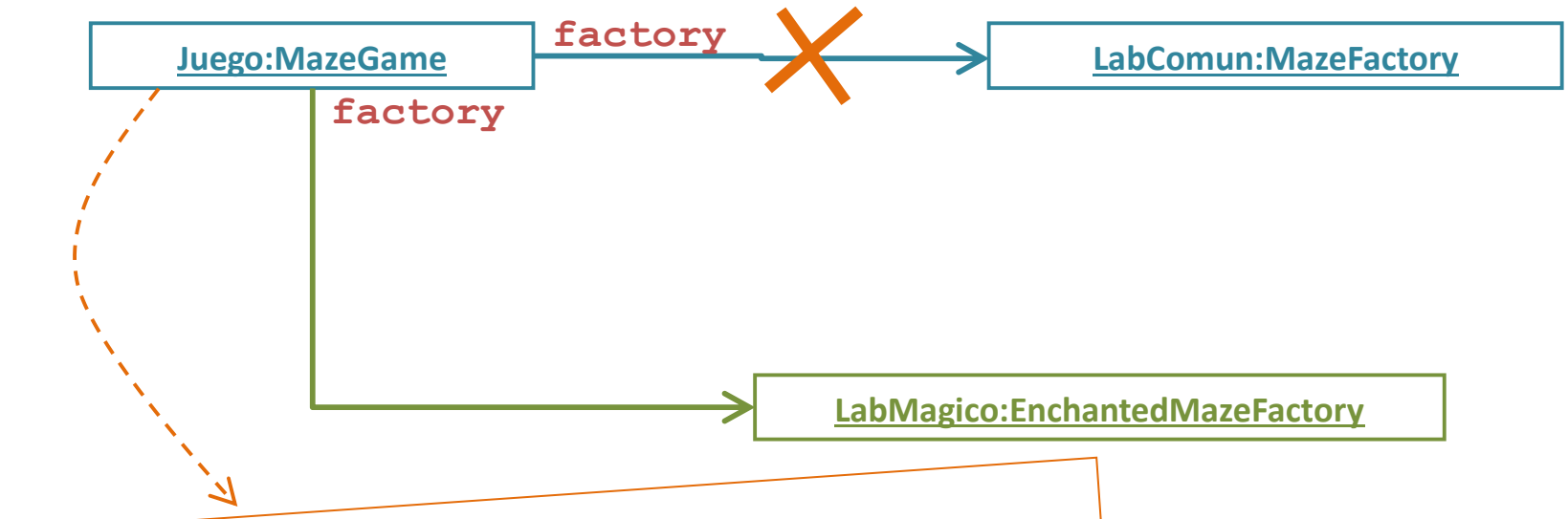
```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);
    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);
    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
    r1->SetSide(West, factory.MakeWall());
    r2->SetSide(North, factory.MakeWall());
    r2->SetSide(East, factory.MakeWall());
    r2->SetSide(South, factory.MakeWall());
    r2->SetSide(West, aDoor);
    return aMaze; }
```

Patrones Creacionales: Abstract Factory

Podemos crear otros tipos de laberintos simplemente utilizando herencia e invocando la operación anterior con el **factory** correspondiente.

```
class EnchantedMazeFactory : public MazeFactory {
   public:
      EnchantedMazeFactory();

      virtual Room* MakeRoom(int n) const {
                  return new EnchantedRoom(n, CastSpell());
      }

      virtual Door* MakeDoor(Room* r1, Room* r2) const {
                  return new DoorNeedingSpell(r1, r2);
      }
   protected:
      Spell* CastSpell() const; };
```

Patrones Creacionales: Abstract Factory




```
…
Room* r1 = factory.MakeRoom(1);
Room* r2 = factory.MakeRoom(2);
Door* aDoor = factory.MakeDoor(r1, r2);
…
```

Patrones Creacionales: Abstract Factory

Otro ejemplo: Concesionaria de Autos

- Implementar un sistema para una concesionaria de autos.
- Supongamos que este sistema se va a correr en diferentes concesionarias (Ford, Fiat, BMW, etc)
- En cada concesionaria crearemos 2 tipos de vehículos (por ejemplo sedán, suv).
- Cuando el sistema corre en una concesionaria:
  - Ford debemos crear Focus o Ecosport
  - Fiat debemos crear Siena o F500x
  - BMW: Serie 3 o X1.

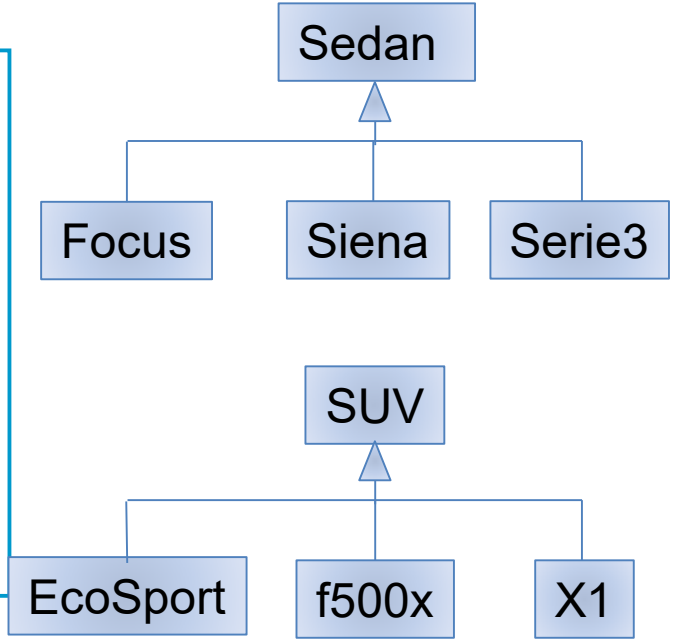
En diferentes corridas del sistema debemos crear objetos distintos

Patrones Creacionales: Abstract Factory

Otro ejemplo: Concesionaria de Autos

Tenemos que abstraer la creación de objetos

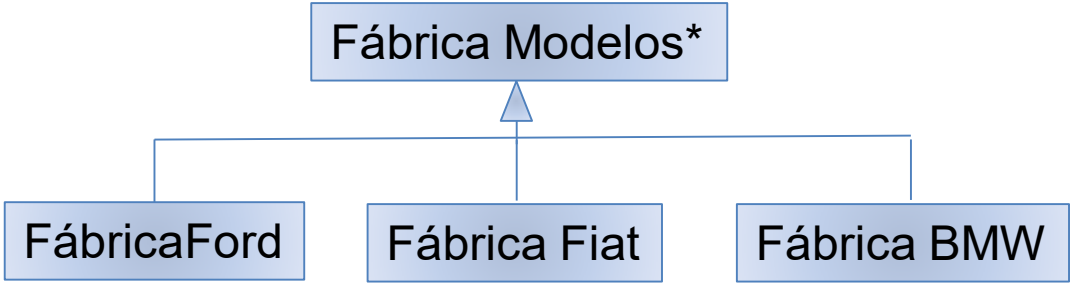
```
FábricaModelos *
 subrutinas:
      *crearSedan(): Sedan
      *crearSUV(): SUV
      *crearCupe(): Cupe
      *crearUtilitario(): Utilitario
      .
      .
      .
```

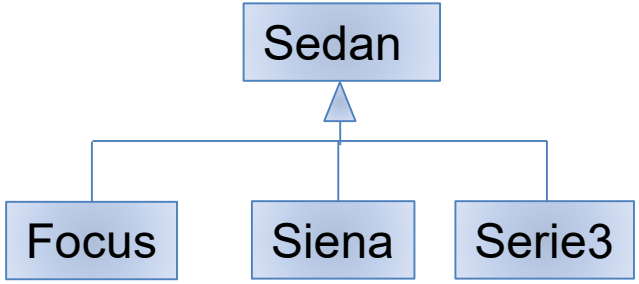


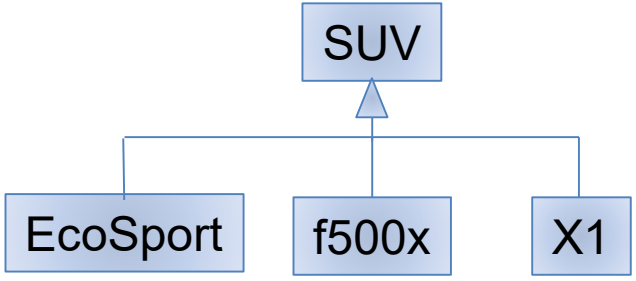
Patrones Creacionales: Abstract Factory

Otro ejemplo: Concesionaria de Autos

Tenemos que abstraer la creación de objetos







Patrones Creacionales: Abstract Factory

Otro ejemplo: Concesionaria de Autos

Tenemos que abstraer la creación de objetos

```
FábricaFord
      Sedan crearSedan(){
            return new Focus();}
      SUV crearSUV() {
            return new EcoSport();}
      SUV crearUtilitario() {
            return new Transit();}
      SUV crearPickUp() {
            return new F150();}
```

Patrones Creacionales: Abstract Factory

Otro ejemplo: Concesionaria de Autos

Desde el código cliente…

```
FabricaModelos fab;
fab= new FabricaFord();
Sedan s;
SUV suv;
Utilitario u;
s = fab.crearSedan();
suv = fab. crearSUV();
u = fab.crearUtilitario();
```

Patrones Creacionales: Prototype

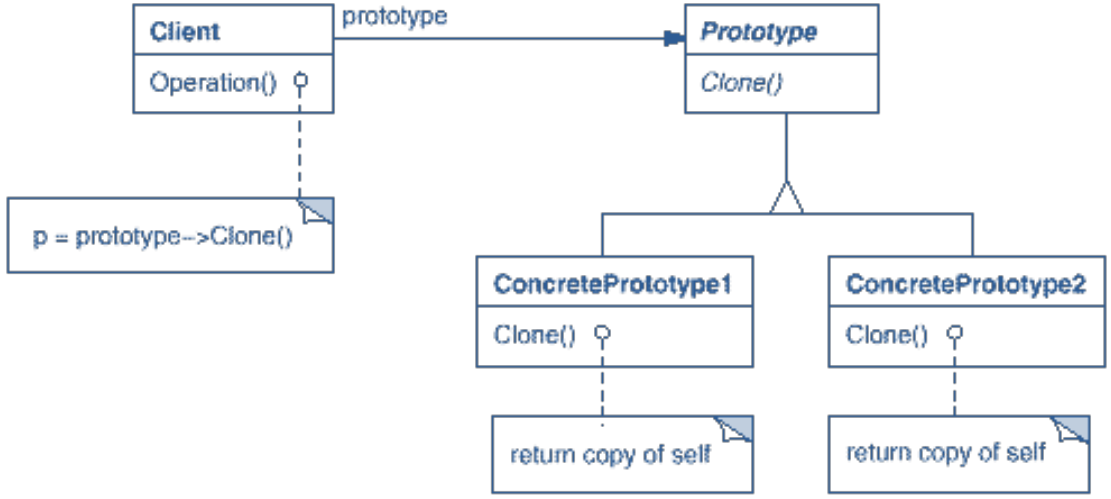
- **Intención:** **especifica los tipos de objetos a crear utilizando una instancia prototipo, y crea nuevos objetos copiando esta instancia.**

- **Aplicabilidad:** Usamos este patrón
  - cuando **las clases a instanciar son especificadas en tiempo de ejecución**, por ejemplo, por carga dinámica.
  - cuando instancias de una clase pueden tener sólo una de muchas combinaciones de estados, lo que puede obtenerse clonando prototipos en lugar de hacerlo manualmente.

Este patrón utiliza "prototipos" de los objetos a crear, los cuales obtiene por clonación.

Patrones Creacionales: Prototype

Estructura:



Participantes:

- **Prototype** declara una interfaz para la autoclonación.
- **ConcretePrototype** implementa una operación para clonarse a sí mismo.
- **Client** crea un nuevo objeto solicitándole al prototipo que se clone a sí mismo.

Patrones Creacionales: Prototype

Apliquemos este patrón al escenario de los laberintos.

```
class MazePrototypeFactory : public MazeFactory {
    public:
        MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

        virtual Maze* MakeMaze() const;
        virtual Room* MakeRoom(int) const;
        virtual Wall* MakeWall() const;
        virtual Door* MakeDoor(Room*, Room*) const;

    private:
        Maze* _prototypeMaze;
        Room* _prototypeRoom;
        Wall* _prototypeWall;
        Door* _prototypeDoor; };
```

El constructor inicializa los prototipos con los parámetros.

Patrones Creacionales: Prototype

Las funciones miembro para crear habitaciones, puertas y paredes simplemente **solicitan clones a los prototipos**.

```
Wall* MazePrototypeFactory::MakeWall () const {
        return _prototypeWall->Clone();
}
Door* MazePrototypeFactory::MakeDoor (Room* r1, Room *r2) const {
        Door* door = _prototypeDoor->Clone();
        door->Initialize(r1, r2);
        return door;
}
```

Para crear laberintos utilizamos la operación **MakeMaze()**

Para crear laberintos con otras características (por ejemplo, habitaciones con ciertas propiedades adicionales), simplemente inicializamos **MazePrototypeFactory** con otros prototipos.

PATRONES ESTRUCTURALES

Structural Patterns

Patrones Estructurales

Los **patrones estructurales** se refieren a cómo **las clases** y **los objetos** son **organizados** para conformar estructuras más grandes.

Pueden ser:

- **patrones de clases**, basados en la utilización de herencia, o
- **patrones de objetos**, basados en la técnica de composición.

La mayor variedad se encuentra en los patrones estructurales de objetos, en donde los objetos se componen para obtener nuevas funcionalidades.

Los objetos **interactúan** entre ellos por medio de sus **interfaces**.

Patrones Estructurales: Adapter

- **Intención**: Convertir la interfaz de una clase en otra interfaz que el cliente espera.
- **Alias**: *Wrapper*.
- **Aplicabilidad**: Usaremos este patrón cuando:
  - Deseamos usar una **clase existente**, pero su interfaz no es la que necesitamos.
  - Queremos crear una clase reutilizable que coopera con otras clases no relacionadas, probablemente con interfaces incompatibles.
  - Necesitamos usar varias subclases, pero no es práctico adaptar sus interfaces heredando de ellas, por lo que apelamos a un **objeto adaptador**. En este caso se utiliza la versión **Object Adapter**

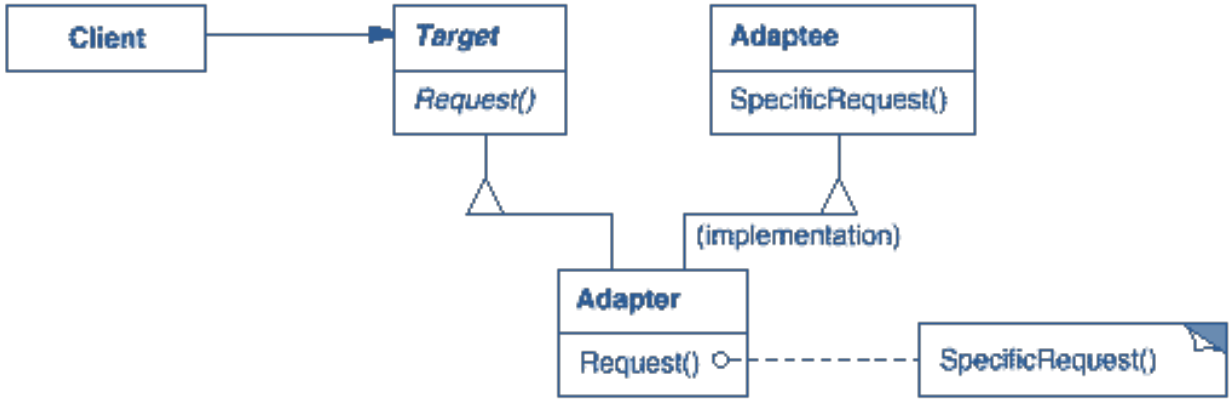
Patrones Estructurales: Adapter

Básicamente, es un **encapsulado de los métodos** y una traducción directa.

Lo usaremos cuando queramos **adaptar una clase con interfaz diferente**, a la interfaz que necesitamos.

Patrones Estructurales: Adapter

Estructura: *(versión Class-Adapter)*

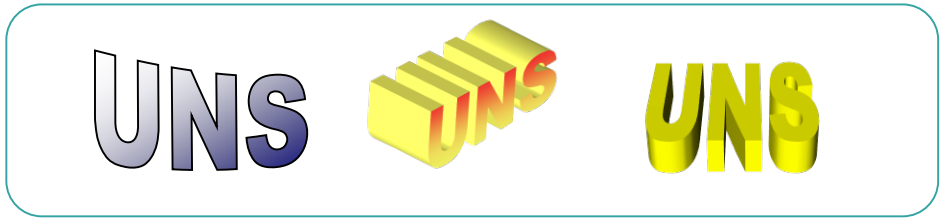


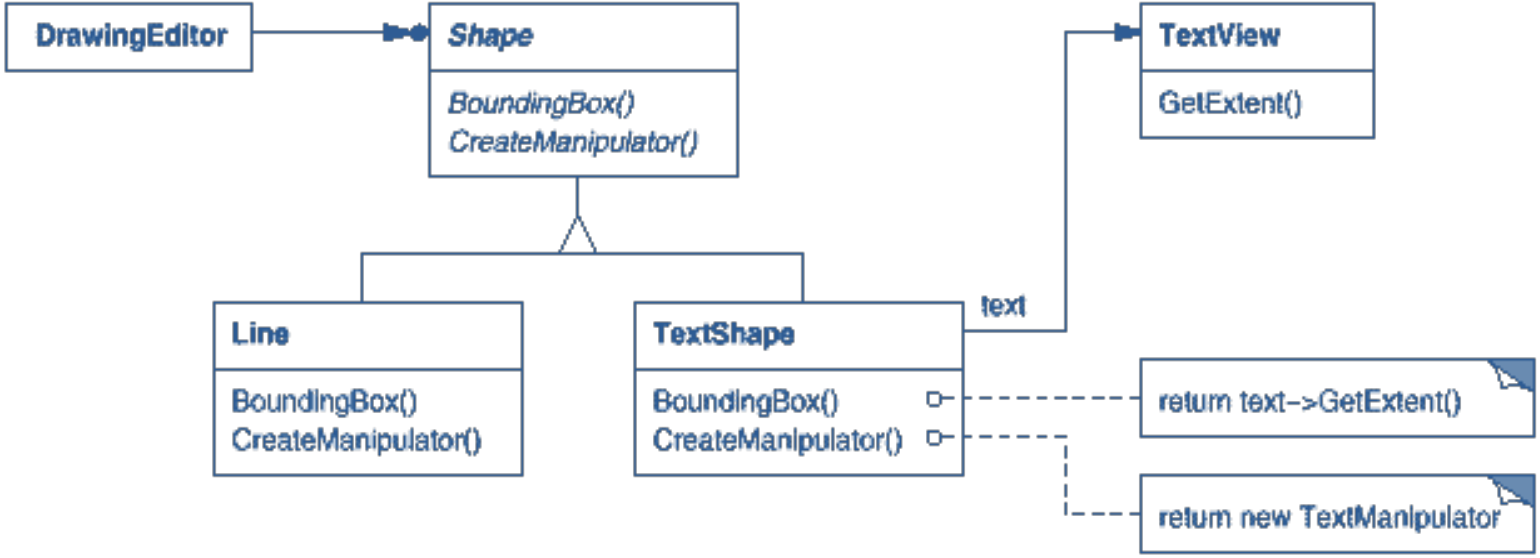
Participantes:

- **Target**: define la interfaz que el cliente usa.
- **Adaptee**: interfaz existente que necesita adaptarse.
- **Adapter**: adapta la interfaz de **Adaptee** a la que necesita el cliente.

Patrones Estructurales: Adapter

Motivación:





Patrones Estructurales: Adapter

```
class Shape {
 public:
  Shape();
  virtual void BoundingBox(Point& bLft, Point& tRight) const;
  virtual Manipulator* CreateManipulator() const;};
```

Interfaz esperada en nuestra aplicación

```
class TextView {
 public:
  TextView();
  void GetOrigin(Coord& x, Coord& y) const;
  void GetExtent(Coord& width, Coord& height) const;
  virtual bool IsEmpty() const;};
```

Clase que queremos usar, pero la interfaz es incompatible.

```
class TextShape : public Shape, private TextView {
 public:
  TextShape();
  virtual void BoundingBox(Point& bLft, Point& tRght) const;
  virtual bool IsEmpty() const;
  virtual Manipulator* CreateManipulator() const;};
```

Clase que adapta **TextView** con la interfaz **Shape**

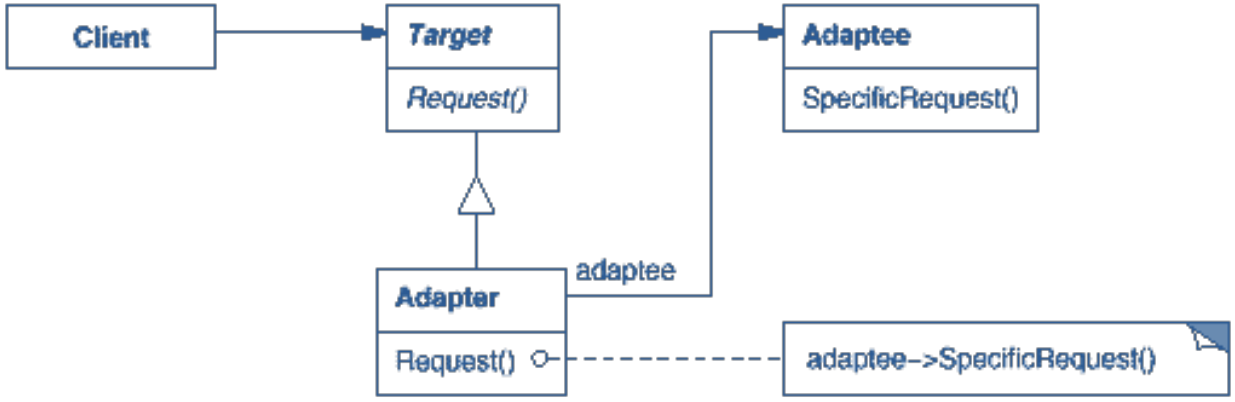
Patrones Estructurales: Adapter

```
class TextShape : public Shape, private TextView {
 public:
  TextShape();
  virtual void BoundingBox(Point& bLft, Point& tRght) const;
  virtual bool IsEmpty() const;
  virtual Manipulator* CreateManipulator() const;
};
```

```
void TextShape::BoundingBox ( Point& bLeft, Point& tRght ) const {
        …
        GetExtent(width, height);
        …
}

bool TextShape::IsEmpty () const {
        return IsEmpty();
}
```

Patrones Estructurales: Adapter

Estructura: *(versión Object-Adapter)*



Participantes:

- **Target**: define la interfaz que el cliente usa.
- **Adaptee**: interfaz existente que necesita adaptarse.
- **Adapter**: adapta la interfaz de Adaptee a la que necesita el cliente.

Patrones Estructurales: Adapter

El **object-adapter** utiliza **composición de objetos** en lugar de herencia múltiple.

```
class TextShape : public Shape {
    public:
        TextShape(TextView*);
        virtual void BoundingBox(Point& bLft, Point& tRight)const;
        virtual bool IsEmpty() const;
        virtual Manipulator* CreateManipulator() const;
    private:
        TextView* _text;
    };
```

Patrones Estructurales: Adapter

Cada operación del **adapter** debe derivar (“adaptar”) las consultas de la interfaz **Shape** en función de la interfaz de **TextView** (interfaz adaptada).

```
virtual void BoundingBox(Point& bLft, Point& tRight)const {
        …
        _text->GetExtent(width, height);
        …
}
```

Patrones Estructurales: Composite

Algunas aplicaciones necesitan manipular a la vez **objetos** y **agrupacione**s de objetos **de manera indistinta**.

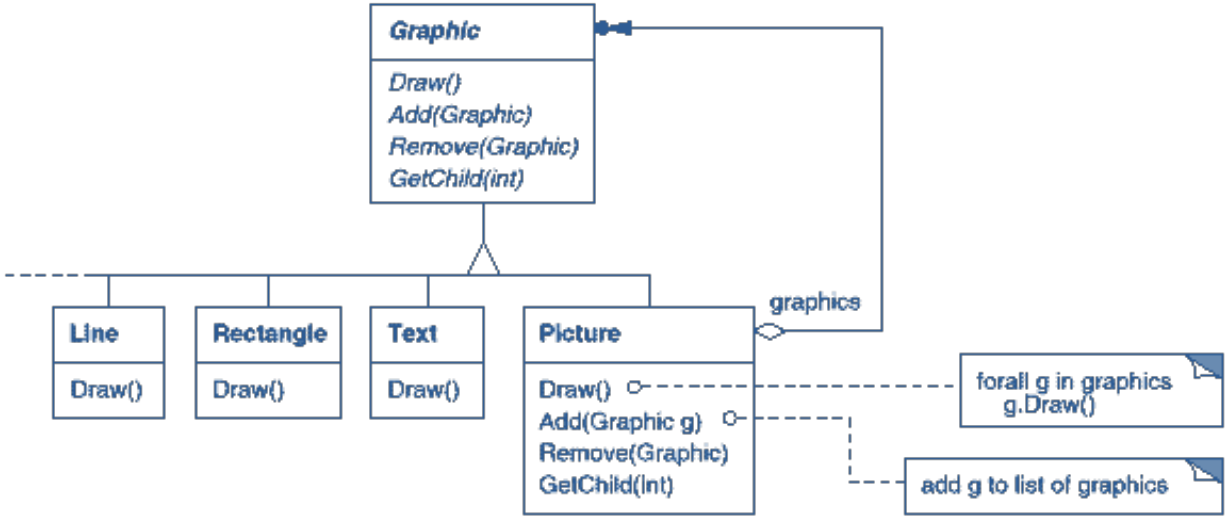
Por ejemplo, una aplicación gráfica necesita manipular líneas, círculos y rectángulos tanto como figuras compuestas por estos elementos, o por otras figuras compuestas.

La aplicación necesita **distinguir** estos objetos, lo cual complica un poco el escenario general.

La clave en este caso es **abstraerse de los dos elementos**, y trabajar con una clase abstracta que represente tanto los objetos primitivos como los objetos compuestos.

Patrones Estructurales: Composite

En el caso de la aplicación gráfica, la organización de clases puede ser:



Esta **composición recursiva** de objetos es la propuesta del patrón de diseño **Composite**.

Patrones Estructurales: Composite

- **Intención**: Compone objetos en estructuras de árbol para representar jerarquías de relación "*es-parte-de*".

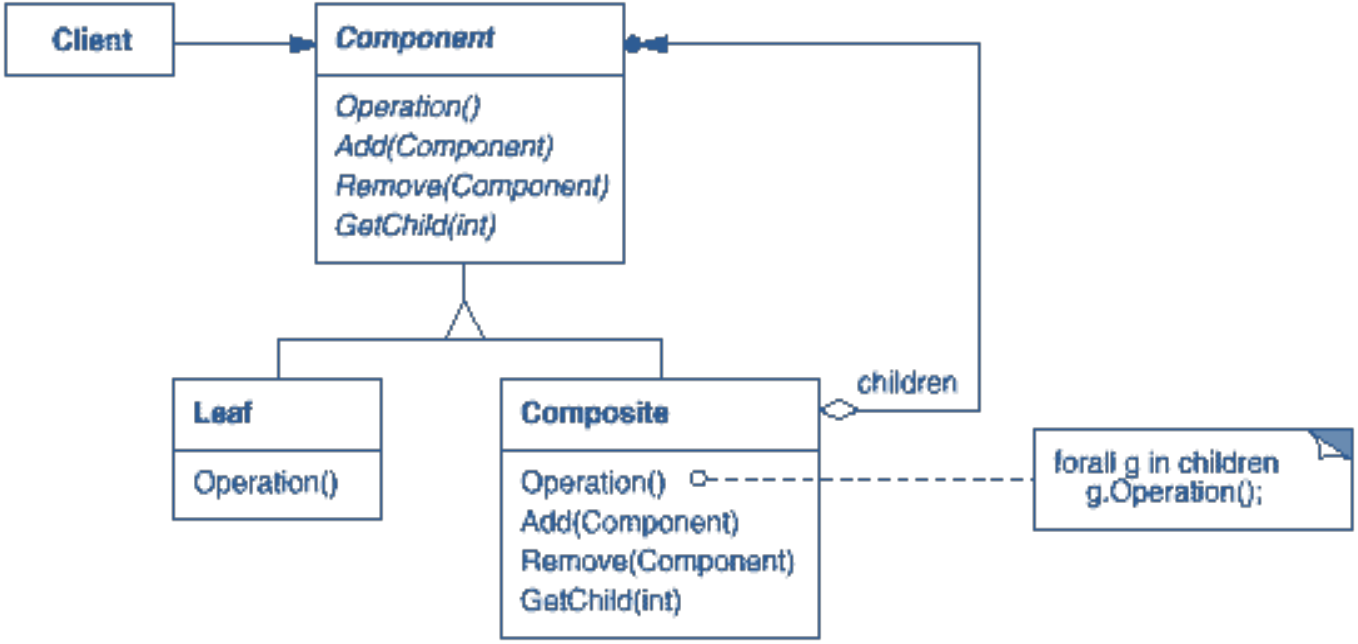
- **Aplicabilidad**: Usaremos este patrón cuando:

    - Queremos representar jerarquías de objetos modelando la relación **"es-parte-de"** (part-whole hierarchies)

    - Queremos que el cliente ignore la distinción entre **objetos compuestos** y **objetos individuales**. El cliente tratará todos los objetos de la estructura compuesta de manera uniforme.

Patrones Estructurales: Composite

Estructura:



Patrones Estructurales: Composite

Participantes:

- **Component**: declara la interfaz para los objetos en la composición. Implementa el comportamiento por defecto para todos los objetos. Declara la interfaz para acceder y manipular los componentes hijos.
- **Leaf**: representa los objetos simples en la composición. Define el comportamiento de tipos primitivos.
- **Composite**: define el comportamiento para componentes compuestos. Almacena estos componentes e implementa operaciones relacionadas a su administración.
- **Client**: Manipula los objetos en la composición

Patrones Estructurales: Composite

```java
public interface ISueldo {
    public double getSueldo();

}
```

```java
public class Empleado implements ISueldo {
    private String nombreCompleto, cargo;
    private double sueldo;
    // y todos los atributos propios del empleado...

    public Empleado(String nombreCompleto, String cargo, double sueldo){
        setCargo(cargo);
        setNombreCompleto(nombreCompleto);
        setSueldo(sueldo);
    }

    public double getSueldo() {
        return sueldo;
    }
```

Patrones Estructurales: Composite

```java
public interface ISueldo {
    public double getSueldo();

}
```

```java
public class Composite implements ISueldo {
    private ArrayList<ISueldo> empleados = new ArrayList<ISueldo>();

    @Override
    public double getSueldo() {
        double sumador = 0;
        for (int i = 0; i < empleados.size(); i++) {
            sumador = sumador + empleados.get(i).getSueldo();
        }

        return sumador;
    }

    public void agrega(ISueldo p) {
        empleados.add(p);
    }

}
```

PATRONES DE COMPORTAMIENTO

Behavioral Patterns

Patrones de Comportamiento

Los **patrones de comportamiento** se centran en los **algoritmos** y la **asignación de responsabilidades** entre los objetos.

Son patrones tanto de **clases** y **objetos** (similares a los anteriores) como de comunicación entre ellos. Caracterizan flujo de control complejo.

Los patrones de comportamiento de **clases** *(behavioral class patterns)* utilizan **herencia** para distribuir el comportamiento entre las clases.

Los patrones de comportamiento de **objetos** *(behavioral object patterns)* utilizan **composición** de objetos en lugar de herencia.

Algunos describen cómo los objetos cooperan entre sí para realizar una tarea compleja, imposible para sólo uno de ellos.

Patrones de Comportamiento: Command

A veces es necesario solicitar servicios que desconocemos sobre objetos que ignoramos.

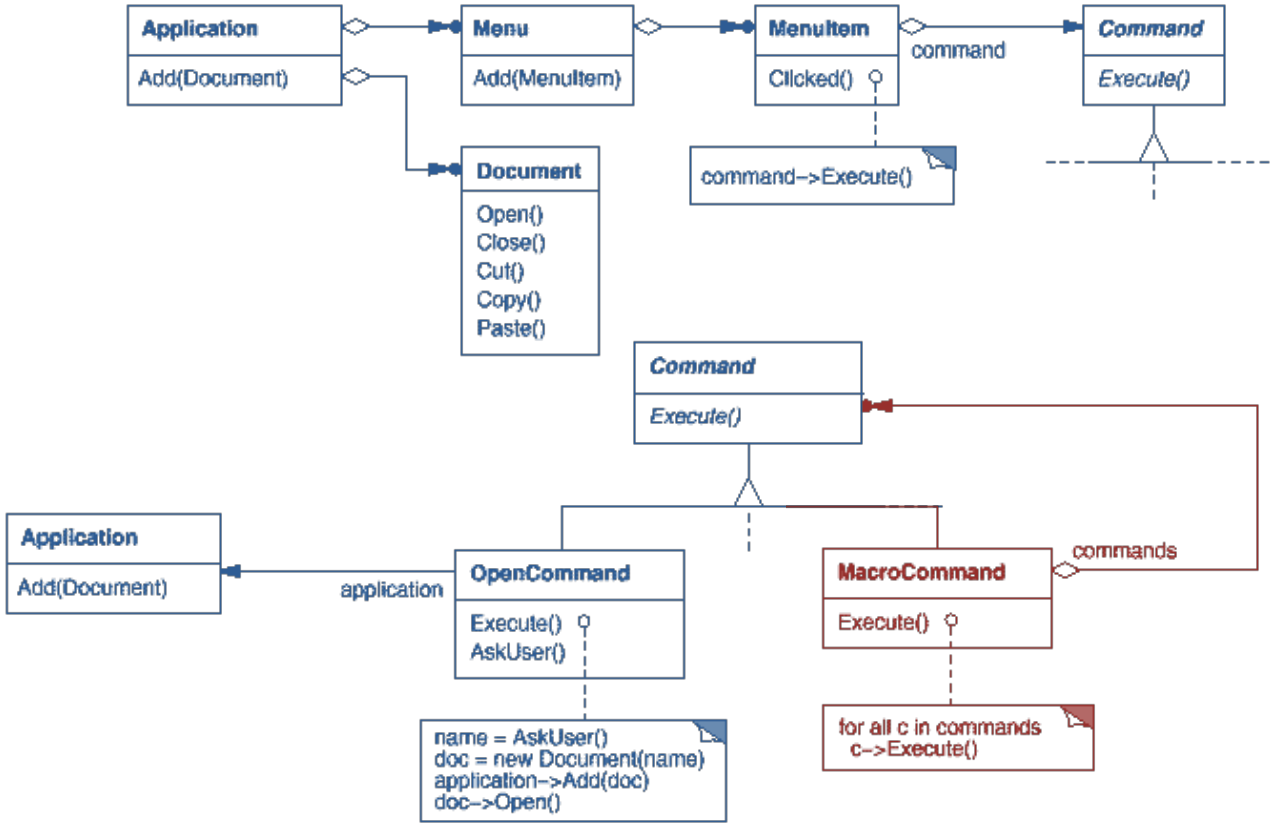
Por ejemplo, por medio de herramientas (toolkits) para implementar menús de usuario, con botones, listas, etc.

Obviamente, una opción del menú no implementa explícitamente la tarea. Sólo la aplicación que usa el toolkit sabe qué debe hacerse en qué objeto.

La idea es que el **pedido mismo sea un objeto**.

Podemos declarar una clase **Command** que declara una interfaz para ejecutar operaciones. Las clases **Command** concretas (descendientes) especifican el receptor de la operación, vinculando el requerimiento con el objeto en cuestión.

Patrones de Comportamiento: Command



Patrones de Comportamiento: Command

Intención: Encapsular el pedido (mensaje, solicitud) como un objeto, permitiendo parametrizar los clientes con diferentes pedidos, encolar o registrar los pedidos y proveer operaciones para deshacer pedidos previos.

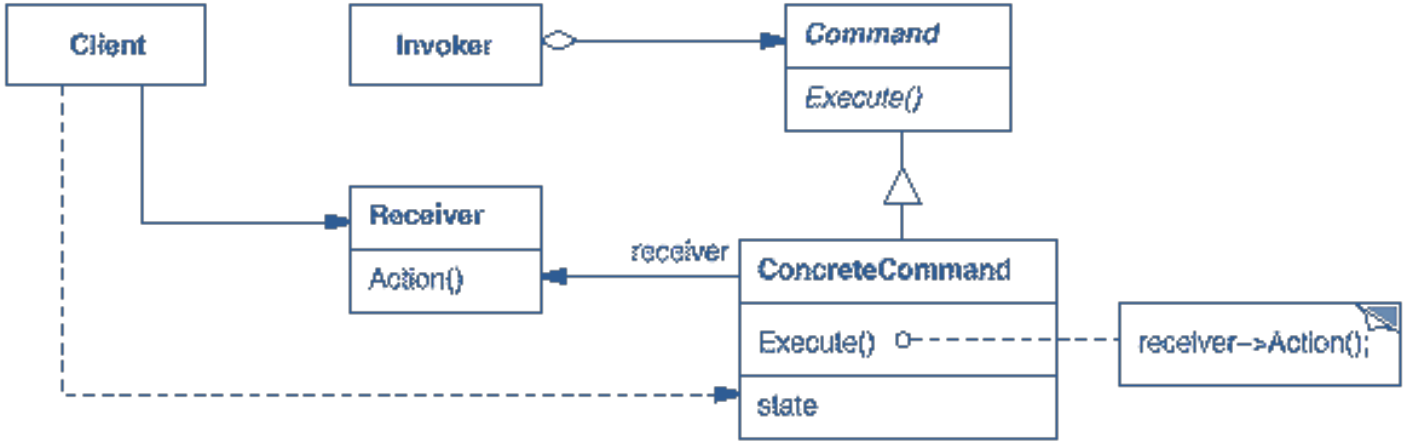
Alias: Action, Transaction.

Aplicabilidad: Usaremos este patrón cuando deseamos:

Parametrizar objetos para una acción a realizar.
Especificar, encolar y ejecutar pedidos en momentos diferentes.
Proveer la posibilidad de deshacer acciones.
Proveer registros de auditoría y salvaguarda.
Estructurar un sistema en función de operaciones de alto nivel construidas en base a operaciones primitivas.

Patrones de Comportamiento: Command

Estructura



Patrones de Comportamiento: Command

Participantes:

Command: declara una interfaz para ejecutar operaciones.

ConcreteCommand: define el vínculo entre el objeto **Receiver** y una acción. Implementa **Execute()** invocando las operaciones correspondientes sobre el Receiver.

Client: crea un objeto **ConcreteCommand** y setea el receptor.

Invoker: solicita al comando realizar su tarea.

Receiver: conoce cómo realizar operaciones asociadas al llevar a cabo un pedido. Cualquier clase puede actuar como **Receiver**.

Patrones de Comportamiento: Command

Ejemplo:

Considérese un "simple" interruptor. En este ejemplo configuramos el interruptor con dos órdenes: encender la luz y apagar la luz.

Un beneficio de esta implementación en particular del patrón **command** es que el interruptor puede ser usado en cualquier dispositivo, no solo con una luz (el interruptor en el siguiente ejemplo enciende y apaga la luz, pero el constructor del interruptor es capaz de aceptar cualquier subclase de **Command** para sus dos parámetros).

Por ejemplo, se podría configurar el interruptor para encender un motor.

Patrones de Comportamiento: Command

```java
/* The Invoker class */
public class Switch {

    private Command flipUpCommand;
    private Command flipDownCommand;

    public Switch(Command flipUpCmd, Command flipDownCmd) {
         this.flipUpCommand = flipUpCmd;
         this.flipDownCommand = flipDownCmd;
    }

    public void flipUp() {
         flipUpCommand.execute();
    }

    public void flipDown() {
         flipDownCommand.execute();
    }
}
```

```java
/* The Receiver class */
public class Light {

      public Light() {  }

      public void turnOn() {
         System.out.println("The light is on");
      }

      public void turnOff() {
         System.out.println("The light is off");
      }
}
```

Patrones de Comportamiento: Command

```java
/* The Command interface */
public interface Command {
    void execute();
}
```

```java
public class FlipUpCommand implements Command {

   private Light theLight;

   public FlipUpCommand(Light light) {
        this.theLight=light;
   }

   public void execute(){
      theLight.turnOn();
   }
}
```

```java
public class FlipDownCommand implements Command {

   private Light theLight;

   public FlipDownCommand(Light light) {
        this.theLight=light;
   }

   public void execute() {
      theLight.turnOff();
   }
}
```

Patrones de Comportamiento: Command

```
public static void main(String[] args){
   // Check number of arguments
   if (args.length != 1) {
      System.err.println("Argument \"ON\" or \"OFF\" is required.");
      System.exit(-1);
   }

   Light lamp = new Light();
   Command switchUp = new FlipUpCommand(lamp);
   Command switchDown = new FlipDownCommand(lamp);
   Switch mySwitch = new Switch(switchUp, switchDown);

   switch (args[0]) {
      case "ON":
         mySwitch.flipUp();
         break;
      case "OFF":
         mySwitch.flipDown();
         break;
      default:
         System.err.println("Argument \"ON\" or \"OFF\" is required.");
         System.exit(-1);
   }
```

Patrones de Comportamiento: Oberver

Un problema recurrente en la programación orientada a objetos es el mantenimiento de la **consistencia** entre **objetos relacionados y cooperativos,** sin padecer de alto acoplamiento.

Este problema cobra también importancia en la representación visual de objetos complejos.

Cuando el objeto cambia, su representación visual (que también es un objeto) debe cambiar acordemente (ejemplo: gráficos en Excel)

Una forma de lograr esto es **organizar los objetos en observadores de un aspecto particular** (*observers-subject*).

La propuesta de organización se refleja en el patrón **Observer**.

Patrones de Comportamiento: Oberver

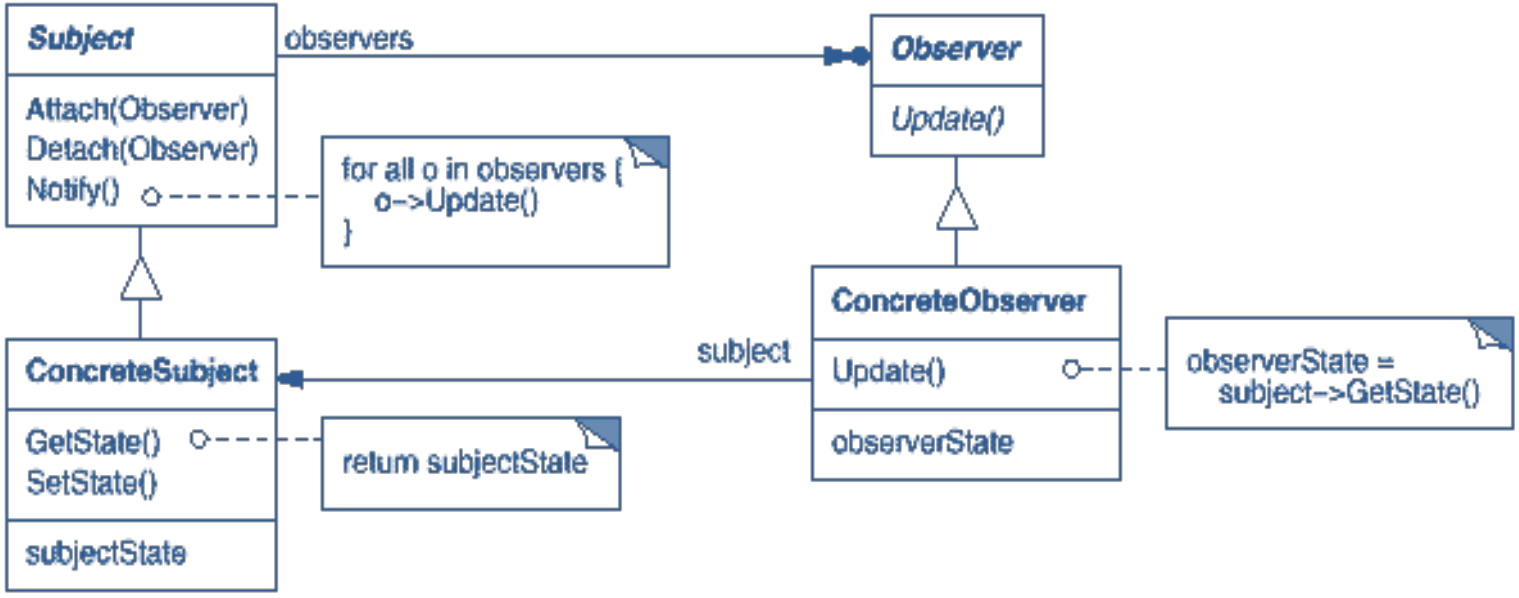
Intención: Define una ***dependencia entre objetos de uno-a-muchos*** de forma tal que cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados acordemente.

Alias: *Dependents, Publish-Suscribe*

Aplicabilidad: Usaremos este patrón cuando:

Cuando una abstracción tiene dos aspectos, uno independiente del otro.
Cuando un cambio a un objeto requiere cambios en otros, y no sabemos cuántos objetos necesitan ser cambiados.
Cuando un objeto debería notificar a otros objetos sin realizar suposiciones de quiénes son esos objetos (evitar el acoplamiento)

Patrones de Comportamiento: Oberver



Patrones de Comportamiento: Oberver

Ejemplo: Una aplicación que a través del botón sumar, suma 1.

Java nos proporciona la clase Observable

```
class Observable {
    public void addObserver(Observer o) {}
    protected void clearChanged() {}
    public int countObservers() {return 3;}
    public void deleteObserver(Observer o) {}
    public void deleteObservers() {}
    public boolean hasChanged() {return null}
    public void notifyObservers() {}
    public void setChanged() {}
};
```

Patrones de Comportamiento: Oberver

La vista implementará la interfaz

Observer

```
interface Observer {
 public void update(Observable o, Object arg);
};
```

```
class Vista extends JFrame implements Observer {
JTextField valor;
JButton sumar;

public Vista(Modelo modelo) {
 // crea e inicializa sus elementos
 ActionListener controlador= new Controlador(modelo);
 sumar.addActionListener(controlador);
 // termina de configurarse
}

public void update (Observable o, Object arg){
 Modelo modelo= (Modelo) o;
 Integer i= new Integer(modelo.obtenerValor());
 valor.setText(i.toString());
}

public void activar(){
 setVisible(true);
}
};
```

Patrones de Comportamiento: Oberver

```
class Controlador implements ActionListener{
 Modelo modelo;

public Controlador(Modelo modeloP){
 modelo= modeloP;
}

public void actionPerformed (ActionEvent e){
 modelo.sumar();
}

};
```

Patrones de Comportamiento: Oberver

```
class Controlador implements ActionListener{
 Modelo modelo;

public Controlador(Modelo modeloP){
 modelo= modeloP;
}

public void actionPerformed (ActionEvent e){
 modelo.sumar();
}

};
```

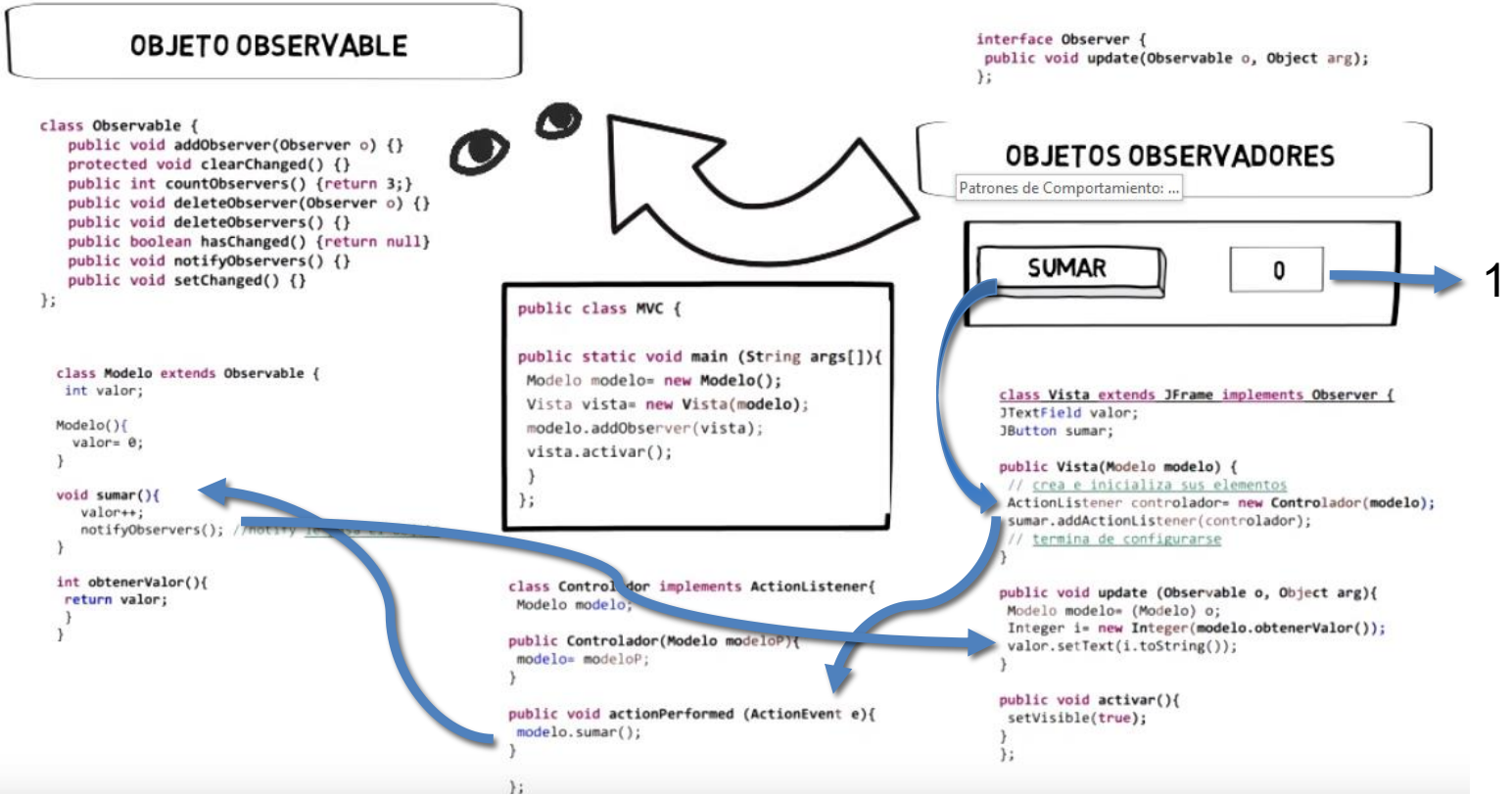
Patrones de Comportamiento: Oberver

```
class Modelo extends Observable {
 int valor;

Modelo(){
  valor= 0;
}

void sumar(){
   valor++;
   notifyObservers(); //notify le pasa el objeto
}

int obtenerValor(){
 return valor;
 }
}
```

Patrones de Comportamiento: Oberver

```
public class MVC {

public static void main (String args[]){
 Modelo modelo= new Modelo();
 Vista vista= new Vista(modelo);
 modelo.addObserver(vista);
 vista.activar();
 }
};
```

Patrones de Comportamiento: Oberver



FUENTE: https://www.youtube.com/watch?v=Z9jX2aWGizI

Material Bibliográfico

- Meyer, B., *Object Oriented Software Construction*. ISE, Inc. 2nd Ed., 1997.

- Abadi, M., & Cardelli, L. A theory of objects. Springer Science & Business Media, 2012.

- Szyperski, C., Component Software: Beyond Object Oriented Programming. AddisonWesley, 2nd Ed., 2011

- Zamir, S., Ed., Handbook of Object Oriented Technology. CRC Press, 2000.